



Macintosh®

Allegro Common LISP

Foreign Function Interface

Overview

The Foreign Function Interface (FFI) lets Allegro CL call functions written in C, Pascal, Assembler, and other languages (such functions are called foreign functions). Foreign functions can, in turn, make calls back to Lisp.

To take advantage of the high-level functions described below, the foreign function code must be compiled into Macintosh Programmer's Workshop (MPW™) object file format. Any compiler which produces object files of the MPW format may be used. Machine code produced by other compilers or assemblers can only be accessed using the low-level function `ff-call`.

The Foreign Function Interface provides two interface syntaxes. The primary syntax is designed specifically for use with Allegro CL on the Macintosh; this is the syntax described by the major portion of this document. The secondary syntax (Franz syntax) was implemented for compatibility with versions of Allegro CL running on other computers; this syntax is described in Franz Inc's *Allegro CL User Guide*, with supplemental notes provided at the end of this document.

Using foreign functions from Allegro CL involves the following steps:

- Write and compile the foreign functions using a compiler which produces MPW object Files.
- Run Allegro CL and load the Foreign Function Interface fasl files.
- Load the MPW object files with the function `ff-load`.
- Define an interface for each foreign function you wish to call. (This is done with `defffun`.)
- Call the foreign functions from Lisp using Lisp syntax.

A call from Lisp to a foreign function looks exactly like a call to another Lisp function. The Lisp function which makes the call (and, for that matter, the programmer) doesn't even need to

know that the function called was written in a different language.

Installation

The files required to use the foreign function interface are found in the folder "Foreign Function Folder." You may wish to move the files "ff.fasl" and "defforeign.fasl" into a directory included in your *module-search-path*. To use the foreign function interface, run Allegro CL and load the file "ff.fasl". If you wish to use Franz syntax, you must also load the file "defforeign.fasl".

Foreign Function Interface Operations

ff-load *files* &key :entry-names :libraries [Function]
 :library-entry-names :ffenv-name
 :replace

loads the MPW object files specified by *files*, and returns a *foreign function environment*.

The foreign function environment returned consists of code segments, a jump table, a static data area, and a collection of active entry point names. Dead code removal is performed so that only code and data reachable from the active entry points will be included in the environment.

<i>files</i>	should be a filename, pathname or a list of filenames and pathnames of MPW object files.
:entry-names	a list of strings naming all the entry points in <i>files</i> which should be active. If :entry-names is not specified or is nil, all entry points in <i>files</i> will be active. <i>Note that these strings are case-sensitive.</i>
:libraries	a list of additional object files to load. These differ from the files in <i>files</i> in that by default, entry points in libraries are not considered active (so that code from libraries will not be included in the link unless needed by other functions).
:library-entry-names	specifies a list of active entry point names in libraries. This overrides the default for libraries of only including entry points used by other functions.
:ffenv-name	should be a symbol. If given and not nil, ff-load will check whether there is already an environment loaded with the given name. If there is, the action taken depends on the :replace argument (described below). If there isn't, the specified files are loaded and the resulting environment is given the name passed in this argument. This argument can be used to make ff-load behave something like require.
:replace	if given and not nil, the files are always loaded and any previously loaded environment of the same name (as specified by the :ffenv-name argument) is disposed of. (The previously existing environment will only be disposed if the loading is successful.)

Each call to ff-load produces a distinct foreign function environment, with its own global space, function code, etc. There is no sharing of code or data between environments produced by separate calls to ff-load. For example, if two different calls to ff-load require a library function "atoi", they will each get their own copy of "atoi". If they each

reference a global variable "errno", they will get their own copy of "errno". To share data and library code between routines, you must link the routines in a single call to `ff-load`.

Example:

```
(def-logical-pathname "MPW;" "Hard Disk:MPW:")
(defparameter *c-libraries* '("MPW;CLibraries:CInterface.o"
  "MPW;CLibraries:StdCLib.o"))
(ff-load "c-hacks;utils.c.o"
  :ffenv-name 'c-utils
  :libraries *c-libraries*
  :library-entry-points
  '("atoi" "strcmp"))

(deffcfun (frob "frob") ...) ;frob is defined in utils.c
(deffcfun (atoi "atoi") ...)
(deffcfun (strcmp "strcmp") ...)
```

dispose-ffenv ffenv

[Function]

disposes of the heap storage used by the environment *ffenv*. If you call a foreign function which is in an environment which has been disposed, you will almost certainly crash.

ffenv a foreign function environment, as returned by `ff-load`. *ffenv* may also be a symbol naming a foreign function environment.

defffun	<i>(Lisp-name entry-name {option value}*)</i> <i>({argspec}*) {result-flag}*</i>	[Macro]
deffcfun	<i>(Lisp-name entry-name {option value}*)</i> <i>({argspec}*) {result-flag}*</i>	[Macro]
deffpfun	<i>(Lisp-name entry-name {option value}*)</i> <i>({argspec}*) {result-flag}*</i>	[Macro]

These macros help you to define a Lisp interface to a foreign function. You describe the arguments the function takes and the result it returns, and `defffun` defines a Lisp function which performs appropriate coercions and type-checks on the arguments and calls the foreign function. `deffcfun` and `deffpfun` are identical to `defffun` except they default the `:language` option (described below, in the section `options`).

<i>Lisp-name</i>	the name of the Lisp function which will be defined by <code>defffun</code> . This must be a symbol.
<i>entry-name</i>	the name of an active entry point in a foreign function environment. It should be a string. <i>Entry point names are case sensitive</i> . If <i>entry-point</i> exists in more than one loaded environment, the environment used is undefined.
<i>option/value</i>	The entry name may be followed by options which further describe the foreign function. These options are described below, in the section <code>options</code> .
<i>argspec</i>	describes a single argument to the foreign function. Argspecs have the general form <i>(lisp-type {flag}*)</i> . They are described in

detail below, in the section **argspecs**.

result-flags

describe the value returned by the function. Result flags are described below, in the section **result flags**.

Options

The *options* provide information on the syntax and calling sequence of the function being defined.

:language the corresponding *value* indicates the language which was used to define the foreign function, which in turn regulates defaults for other options. This option is currently used in the macroexpansion of `deffcfun` and `deffpfun`. Future extensions will support other language types.

:check-args if the corresponding *value* is non-`nil` (the default), the function will perform run time checks to ensure that the actual argument types match the declared expectations. If the *value* is `nil`, this type checking is skipped. This option may be overridden by individual *argspecs*.

:reverse-args if the corresponding *value* is non-`nil`, the arguments are pushed on the stack in reverse order from that specified in the *argspec* list. This is the default if the language is C.

Argspecs

The *argspecs* give information about each argument the foreign function will receive. This includes the Lisp type to expect, and a series of flags. The flags give information on the foreign type, the argument passing method, and whether to perform argument checking.

lisp-type is any valid Lisp type specifier. It declares that the corresponding argument to the Lisp function will be of that type. If argument checking is requested, a `check-type` form will be included in the Lisp function. In addition, *lisp-type* is used to select the argument passing convention and foreign type if not explicitly specified. If *lisp-type* is a symbol (not a list) and there are no *flags*, the parentheses around the *argspec* may be omitted.

flags specify the format (foreign type) in which the corresponding argument should be passed, the method for passing, and whether or not type-checking should be performed on the argument.

The following flags describe the format (foreign type) of the argument to be passed to the foreign function. They are mutually exclusive, that is, you may choose only one of the possible values:

:long The foreign function is expecting a long word value. The Lisp argument should be an integer or a character (in which case its `char-code` is used). The low 32 bits of the value constitute the foreign argument. If the Lisp value is known to be a fixnum (i.e. if *lisp-type* is a subtype of `fixnum` or `character`), the code generated

will be much more efficient.

:word	The foreign function is expecting a word value. The Lisp argument should be an integer or a character (in which case its char-code is used). The low 16 bits of the value constitute the foreign argument. If the Lisp value is known to be a fixnum (i.e. if <i>lisp-type</i> is a subtype of <i>fixnum</i> or <i>character</i>), the code generated will be much more efficient.
:double	The foreign function is expecting a floating point value in the machine double format (8 bytes). The Lisp argument should be a float.
:extended	The foreign function is expecting a floating point value in the machine extended (10-byte) format. The Lisp argument should be a float. :extended is the default float type for C and Pascal.
:ptr	The foreign function is expecting a longword value. The Lisp argument will be passed unmodified. The argument should generally be a pointer to a location on the Macintosh heap, or to a stack block. Passing a pointer to Lisp data is not safe, as such data can be relocated at any time.
:Lisp-ref	The foreign function is expecting a pointer to a Lisp value. A temporary location is reserved (for the duration of the call) in non-relocatable memory, a pointer to the Lisp argument is placed in that location, and the address of the location is passed to the foreign function. The foreign function can access the Lisp data using double indirection just as it would a Macintosh handle. The contents of the location will be updated whenever the Lisp data is relocated.
:cstring	The foreign function is expecting a null-terminated string. The Lisp argument should be a Lisp string. The foreign function must not modify any locations beyond the end of the string. The string may be any length.
:pstring	The foreign function is expecting a Pascal string (a string preceded by a count byte). The Lisp argument should be a string. The foreign function must not modify any locations beyond the end of the string. If the argument has a length of more than 255 characters, an error will be signalled.
(:cstring size)	The foreign function is expecting a null-terminated string in a buffer of <i>size</i> bytes (not including the null). The Lisp argument should be a string of <i>size</i> characters or less. The foreign function must not modify any locations beyond the end of the buffer. <i>size</i> must be a compile-time constant.

(:pstring size) The foreign function is expecting a string preceded by a length byte in a buffer of *size* bytes (not including the length byte). The Lisp argument should be a string *size* characters long or less. The foreign function must not modify any locations beyond the end of the buffer. *size* must be a compile-time constant which is 255 or less.

If no foreign format flag is specified, a default is chosen based on the *lisp-type* and the *:language* option, according to the following table. If *lisp-type* is not listed in the table, there is no default (and the foreign type must be specified).

Language Lisp Type	C	Pascal
Integer	:long	:word
Character	:long	:word
String	:cstring	:pstring
Float	:extended	:extended

Foreign Type Defaults

The following *flags* are used to specify the argument passing method. They are mutually exclusive, that is, you may choose only one of the possible values.

- :by-value** The argument itself is passed to the foreign function (pushed on the stack). This method may not be used to pass strings (i.e. the argument format must not be *:cstring* or *:pstring*).
- :by-address** The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). If the foreign function modifies the argument, the changes will not be visible to Lisp. Use *:by-reference* if you want Lisp to see changes.
- :by-reference** The address of a location containing a copy of the argument is passed to the foreign function (pushed on the stack). In addition, arrangements are made so that upon return from the foreign function, any changes in the copy are reflected in the Lisp argument itself. Thus foreign functions may destructively modify Lisp data structures. Only floats and strings may be passed by reference (i.e. the argument format must be *:cstring* *:pstring*, *:double* or *:extended*). When a string is passed by reference and the foreign function changes the size of the string, an error will be signalled unless the Lisp string is adjustable. *:by-reference* arguments are equivalent to Pascal var arguments.

When no passing method is specified, the default is to pass *:long*, *:word*, *:ptr*

and `:Lisp-ref` arguments by value and others by address. In addition, if the language is C, `:extended` arguments are passed by value.

The following flags specify whether type-checking should be performed on the *lisp-type*. Using one of these flags lets you override the `:check-args` option for the function as a whole. The two flags are mutually exclusive.

- `:check-arg` the argument will be checked at runtime to ensure it is of type *lisp-type*.
- `:no-check-arg` the argument will not be type-checked at runtime.

Result Flags

The value returned by the foreign function is described by the *result-flags*. These flags describe the type and location of the return value.

The type of the return value is described by one of the following keywords. The type determines how the result is coerced before it is passed back to Lisp.

- `:full-long` the result is interpreted as a 32-bit signed integer and returned as a Lisp integer (fixnum or bignum depending on the magnitude). `:full-long` is significantly less efficient than `:long`, as it checks for overflow, and optionally creates a bignum to hold the return value.
- `:long` The return value is a longword. It is truncated to 31 bits and returned as a fixnum. This is the default if the language is C. `:long` is significantly more efficient than `:full-long`.
- `:word` The return value is a 16-bit word. It is interpreted as a signed integer and returned as a fixnum.
- `:double` The foreign result is a double float. It is coerced to a Lisp float.
- `:extended` The foreign result is a float in extended format. It is coerced to a Lisp float.
- `:float` This is a synonym for `:extended`.
- `:char` The foreign result is a character. The low 8 bits of the return value location are interpreted as a character code and returned as a Lisp character.
- `:ptr` The return value is a longword. It is returned unmodified, and should be a pointer to a location on the Macintosh heap.
- `:novalue` The foreign function returns no value. The Lisp function will return `nil`. This is the default if the language is Pascal.

The location of the return value is described by one of the following keywords:

- `:d0-:d7` The foreign function returns the value in the specified data

	register. The default is :d0 if the language is :c.
:a0-a4	The foreign function returns the value in the specified address register.
:stack	The foreign function returns the value on top of the stack.

Short Example:

Assume the file "test.c" contains the following C function:

```
#include <ctype.h>
#include <memory.h>

digitval (ch)
char ch;
{
    if isdigit(ch) return ch - '0';
    else return -1;
}
```

After compiling this file in MPW, we could use it from within Lisp as follows:

```
? (ff-load "test.c.o" :name 'test
      :libraries *c-libraries*)
#<a ff-env>
? (deffcfun (digit-value "digitval") (character)
      :long)
digit-value
? (digit-value #\7)
7
? (digit-value #\A)
-1
```

Low-level Functions

The following low-level functions are used to implement the higher-level functions described above. You may want to use the low-level functions if you require increased speed or flexibility. `ff-call` is faster because it is open coded and the arguments are not coerced or type-checked. This gives more flexibility at the cost of responsibility. Pass the right types or the system will probably crash!

`ff-call` *pointer* {*type-keyword argument*}* [Function]
 [:return-block *pointer* | *return-value-keyword*]

transfers control to the address *pointer*, passing arguments according to the *type-keyword* / *argument* pairs. `ff-call` returns a value according to the :return-block or *return-value-keyword*.

`ff-call` is useful if you have CODE resources with entry points at known offsets. Keep in mind, however, that no type-checking is performed on the arguments, so crashes will occur if bad arguments are passed.

<i>pointer</i>	a pointer to the entry point of the foreign function to be called.
<i>type-keywords</i>	indicate the type of coercion to be performed on the subsequent

	argument and where to place the argument. The possible <i>type-keywords</i> are:
:word	<i>argument</i> (which should be a fixnum) is truncated to 16 bits and the resulting word value is pushed on the stack.
:long	<i>argument</i> (which should be a fixnum) is sign-extended from 31 to 32 bits and the resulting long word value is pushed on the stack.
:ptr	<i>argument</i> is pushed on the stack unmodified as a 32-bit pointer. This should generally be a pointer to the Macintosh heap or stack. It should not be a pointer to the Lisp heap.
:d0-d7	<i>argument</i> (which should be a fixnum) is sign-extended from 31 to 32 bits and placed in the indicated data register.
:a0-a4	<i>argument</i> is placed unmodified in the indicated address register.
:a5	<i>argument</i> is placed unmodified in the A5 register and additionally the new A5 value is stored in the Macintosh low memory global CurrentA5 for the duration of the call.

The stack-based arguments are pushed on the stack in the order in which they appear in the call.

<i>return-value-keyword</i>	indicates the type of value the function returns. If <i>return-value-keyword</i> is not supplied, :novalue is assumed. The following keywords are recognized:
:word	A 16-bit result is read from the top of the stack, sign-extended and returned as a fixnum.
:long	A 32-bit result is read from the top of the stack, truncated to 31 bits and returned as a fixnum.
:ptr	A 32-bit result is read from the top of the stack and returned unmodified.
:d0-d7	The value of the indicated data register is truncated to 31 bits and returned as a fixnum.
:a0-a4	The value of the indicated address register is returned unmodified.
:novalue	No value is returned from the function. ff-call will return nil.

If the return value is to be taken from the stack, room is left on the stack for the return value

before any stack-based arguments are pushed. The foreign function is entered at *pointer* with the return address on the top of the stack. The function does not have to obey strict stack discipline, but it must never lower the stack beyond its arguments (i.e. it should never pop anything more than its arguments, and it should not modify any data on the stack beyond the arguments).

:return-block If this keyword appears in a call to `ff-call`, it should be followed by a pointer to a block of memory where the return values will be placed. This mechanism lets a foreign function return values from multiple registers and positions on the stack.

If `:return-block` is specified, then the `return-spec` in `ff-call` may be a list of `:a0-:a4, :d0-:d7, :word, :long`. The values are placed in the return block in the order specified, totally unmodified. For example:

```
(%stack-block ((answer 10))
  (ff-call fn :return-block answer :a0 some-arg
    '(:a0 :word :d0))
  (format t
    "~&Fn returned: A0=~X,
    Top-word-on-stack=~S,D0=~X"
    (%get-long answer 0)
    (%get-word answer 4)
    (%get-long answer 6)))
```

It is the user's responsibility to make sure the `:return-block` has room for all the values specified in the return specification.

Here's how to call `digitval` (see previous example) using `ff-call`.

```
(= (multiple-value-bind (entry a5)
    (ff-lookup-entry "digitval")
    (ff-call entry :a5 a5 :long (char-code #\7) :d0))
  7)
```

ff-lookup-entry entry-name

[Function]

Returns two values describing the entry point named *entry-name*. The entry point must be an active entry point in some previously loaded environment. Note that entry point names are case-sensitive. The first value returned is a pointer to the entry point. The second value is the `a5` pointer for the environment where the entry point was found. If *entry-name* does not exist, `nil` is returned. If *entry-name* exists in more than one loaded environment, the specific environment returned is undefined.

entry-name a string giving the name of an entry point in a foreign function environment. The string is case-sensitive.

Examples:

```
;frob is a foreign function:
(multiple-value-bind (entry a5) (ff-lookup "frob")
  (ff-call entry :a5 a5))
;FrobCount is a static integer variable:
(format T "There are ~D frobs."
  (%get-long (ff-lookup-entry "FrobCount")))
```

`ff-lookup-entry` is a relatively slow operation. You would normally call it once at load time and store the results in some easily accessible place, rather than calling it every time you need to reference the entry point. Note that the values returned by `ff-lookup-entry` are pointers, and thus cannot be maintained across dumps and restarts. When restarting a dumped image, you must reinitialize all entry information by recalling `ff-lookup-entry`.

`%word-to-int fixnum`

[Function]

Sign-extends the low word of *fixnum* to a full integer value.

For example:

```
(= (%word-to-int #xFFFF) -1)
(= (%word-to-int 1) 1)
```

`%copy-float float`

[Function]

Returns a copy of *float*, that is a newly consed float which is `eq1` but not `eq` to *float*. The argument can be a Lisp floating point number or a Mac pointer to a memory location which contains a double float in machine format (see 68881 or SANE documentation for details).

Calling Lisp from Foreign Functions

A foreign function may call a Lisp function, receive a returned value, and do further processing before returning to Lisp. A Lisp function to be called by a foreign function must be defined with one of the following macros. The macros create a pointer, which can then be passed to the foreign function.

`defccallable` is used to define Lisp functions that can be called from C. `defpascal` is used to define Lisp function that can be called from the Macintosh toolbox or from user-written Pascal code. Both of these macros put a Lisp function in the function cell of a symbol and a pointer to the C or Pascal entry point in the value cell of a symbol.

A maximum of 62 C-callable and Pascal-callable functions may be defined.

`defccallable name` (((*argument type*))* (:result *return-type*))
[*doc-string*] {*form*}*

[Macro]

`defpascal name` (((*argument type*))* (:result *return-type*))
[*doc-string*] {*form*}*

[Macro]

`defpascal name` ((*argument :reg*)) [*doc-string*] {*form*}*

[Macro]

The syntax is similar to `defun`, except that the lambda list contains pairs rather than symbols, and ends with a type specifier for the value returned by the procedure. Each pair is the name of the argument followed by the argument's type. &optional, &keyword, &rest, and &aux arguments are not permitted because they are not supported by the C or Pascal calling sequence.

Each *type* must be one of `:word`, `:long`, or `:ptr`. `:words` and `:longs` will be passed to the procedure as fixnums, and `:ptrs` will be passed to the procedure as pointers.

return-type specifies the type of the value returned to the calling C or Pascal code. It should be `:word`, `:long`, `:ptr` or `:void`. `:void` is used when the procedure does not return a value. (Omitting *return-type* is equivalent to a *return-type* of `:void`.)

The value cell of *name* will be set to a pointer to the procedure. This pointer may be passed to traps or to C or Pascal code that expects a pointer to a function.

defpascal has an alternative format in which only one *argument* is given, along with the keyword *:reg*. This should be a pointer to a record of type *:regbuf*. When the function is called, the values of the machine registers are copied into the record. The values of the record can be accessed and set with *rref* and *rset*. For a description of records, see the chapter Records, in the Allegro CL User Guide.

The following procedure could be passed to `_TrackControl` and used to track scroll bars:

```
(defpascal my-track-proc ((my-control :ptr)
                          (partCode :word))
  (_SetCtlValue
   :ptr my-control
   :word (+ (_GetCtlValue :ptr my-control :word)
            (case partCode
              (20 -1) ;back one line
              (21 1)  ;forward one line
              (22 (- *page-height*)) ;back one page
              (23 *page-height*))))) ;forward one page
```

It could be used as follows:

```
(if (logbitp 8 (_TrackControl :ptr the-control
                              :long mouse-point
                              :ptr my-track-proc :word))
    (update-window))
```

The following example uses `defccallable`. We use a C function, `addthree`, which takes two arguments, an integer and a pointer to a Lisp function. The C function adds one to its first argument, then calls the Lisp function pointed to by its second argument. This Lisp function is passed the incremented first argument.

The Lisp function (in this case) increments its argument and returns it, whereupon the C function increments it again and returns the value. The flow-of-control here is from Lisp to C to Lisp to C and finally back to Lisp.

`Addthree` is only accurately named if it is passed a pointer to a Lisp function which takes a fixnum argument increments it, and returns the incremented value.

- C code

The syntax for handling pointers to functions is described in "The C Programming Language", Kernighan & Ritchie, 1978, page 209.

```
int addthree (i, Lispfn)
  int i, (*Lispfn) ();
  {
    i = i + 1;
    i = (*Lispfn) (i);
    return i + 1;
  }
```

• Lisp code

`Defccallable` sets the value of its first argument to the entry point for the function that it defines.

```
? (defccallable add-one ((i :long) (:result :long))
  (+ i 1))
#<A Mac Non-Zone Pointer at Address #x4DB054>
```

The pointer to the Lisp function is not a Lisp data type, so use `t` as the type specifier.

```
? (deffcfun (add-three "addthree")
  ((fixnum :long) (t :ptr))
  :long)
ADD-THREE
```

The pointer to the Lisp function is stored in the value cell of `add-one`, so we don't need to quote the symbol or use the special form function.

```
? (add-three 5 add-one)
8
```

Franz Syntax

The Foreign Function Interface provides a syntax option similar to the one used by the Franz Foreign Function Interface. This syntax should only be used by programmers who wish to write code which is portable to Franz systems.

The central operator of the Franz foreign function interface is `defforeign`. Documentation can be found in the Franz Allegro CL User's Manual.

The following features of `defforeign` are not supported on the Macintosh. These features are missing because of differences between the Macintosh and Unix operating systems, and underlying differences in the Foreign Function Interface implementation.

- `:single-float` argument type and return type
- Passing arrays (other than strings) between Lisp and foreign functions
- `:language :fortran`
- `:address` and `:remember-address` arguments.
- The functions `get-entry-point`, `remove-entry-point`, `reset-entry-point-table`, `register-value`, `register-function`, `Lisp-value`, `Lisp-call`, `defun-c-callable`, `foreign-argument`.

As described above, there is no sharing of code and data between separately loaded files. There are no UNIX system libraries; libraries are simply object files, which must be specified by pathname. The Lisp does not support the C runtime system (e.g. standard I/O, etc.).

Extended Example

Following is an expression by expression example of how to use the Foreign Function Interface with C.

You may find it convenient to use Multifinder so that you can quickly switch between MPW and Allegro CL.

- Boot MPW.
- Edit your foreign language code. The examples below use a set of C functions defined in the file "example.c" in the "Foreign Function Folder."
- Compile your code. Use the MPW Build facility to make sure you get all of the right library files.
- Test your code in MPW. This stage isn't strictly necessary, but will ensure that you pass Lisp proven working code. If you don't test your code within MPW, it isn't necessary to link it. Allegro CL only needs the object files.
- Boot Allegro CL and load the Foreign Function Interface
- Lisp code for loading the foreign function files and defining a Lisp interface to the functions is given in the file example.lisp.
- Lisp code for loading the foreign function files and defining an interface to the functions with Franz syntax is given in the file example.franz.
- Call the foreign functions from Lisp. Examples for testing the code are contained in the file example.test.